Arcade.xyz A-1

Security Audit

Apr 25th, 2022

Version 1.0.0

Prepared by

0xMacro.com

macro

Index

Introduction

This document includes the results of the security audit for Arcade.xyz's smart contract code as found in the section titled 'Source Code'. The security audit was performed by the Macro security team from Mar 2, 2022 to Apr 15, 2022.

Arcade.xyz is, at the time of this audit, already live and in use. The contracts being reviewed in this audit are not from the same commit and do feature some differences from those currently deployed to mainnet.

The purpose of this audit is to review the source code of certain Arcade.xyz Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety. Issues below are noted with consideration both towards the similar live codebase as well as to the ongoing development of V2 of the product.

During the course of the audit, an anonymous white hat security researcher notified Arcade.xyz of a previously undiscovered live critical vulnerability. We proceeded to assist Arcade's dev team to craft and execute a recovery plan.

As part of this recovery plan, an updated version of AssetWrapper.sol was deployed, featuring a fix discussed before that point as well as mitigations for the critical vulnerability.

Over the course of the following weeks, the Arcade.xyz team has informed us that ~95% of the at-risk assets have been moved to an updated version of the vulnerable contract. As of this writing, the remaining assets are still pending the Arcade.xyz team getting in contact with asset holders, or loan completion. A system to identify these asset holders, should they visit the website, is in place, and the issue and migration has not yet been otherwise publicly announced so as to protect these asset holders.

Overall Assessment

Arcade.xyz's codebase conforms to many best practices and design patterns, and reflects knowledgeable and modern solidity development. It uses Open Zeppelin libraries correctly, and shows consideration for both security concerns as well as gas efficiency in many implementation details.

That said, we have identified a few low severity issues, and a few areas for improvement including preventative use of some defensive patterns, and more careful attention to comments and error messages. There are also a few possible gas optimizations that could be considered. During the course of this audit, we have shared findings with the Arcade.xyz team as we have performed the audit, and discussed improvements when applicable that can be made in the V2 release that is concurrently under development.

Disclaimer: While Macro's review is comprehensive and has surfaced some changes that we recommend, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Slack with the Arcade.xyz team
- The official website docs, and more primarily the github docs

Source Code

The following source code was reviewed during the audit:

Repository	Commit
Github	3de0af1a9ca18aa67efce0f2c953bdaa6d94cc77

Specifically, we audited the following contracts:

Contract	Sha256
AssetWrapper.sol	65A2259506E1A4A3B5CD026D495CF29618CD ACE59EB1612CB186E71C22B4085C
FeeController.sol	10F18C4AD8D05C511A44CBDE7E044B0F0CFD 7E3EA63CF976395C79D23DDD375F
LoanCore.sol	436D2FAC6EAE59EE3DBB4536496C19437C9FB 18D17661977FFDA0CAECCC921C3
FlashRollover.sol	F7238ACCB950E353CC4E7A0BC02DFB32BE5B 5967E87DB7B707B3C505884C198E
OriginationController.sol	86FFE17C24E3CE78ACFE38311A1821881644C CD11F99BF012706B2811E04EDDE
RepaymentController.sol	9A4C34746DF439D901F14997E920BF0ED0303 AFA96D2D4554C898E618011EA19
PromissoryNote.sol	4E2A38C4F8E2117D1A6D91923BEDE969920A ABF64264BB82377BFE91B8539984
PunkRouter.sol	7FBA5755E3F0C5DFD7DFD7FB8F6E9BF0B966E 769ACDD0A6D6512CEB02139FED7

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Methodology

The audit was conducted in several steps.

First, we reviewed in detail all available documentation and specifications for the project, as described in the ‘Specification’ section above. We also conducted a survey on comparable competitor product designs.

Second, we performed a thorough manual review of the code, checking that the code matched up with the specification, as well as the spirit of the contract (i.e. the intended behavior). During this manual review portion of the audit we primarily searched for security vulnerabilities, unwanted behavior vulnerabilities, and misaligned incentives.

Third, we performed the automated portion of the review consisting of measuring test coverage (while also assessing the quality of the test suite) and evaluating the results of various symbolic execution tools against the code.

Lastly, we performed a final line-by-line inspection of the code – including comments –in effort to find any minor issues with code quality, documentation, or best practices.

Issues Descriptions and Recommendations

Severity Level Reference

Level	Description
High	The issue poses existential risk to the project, and the issue identified could lead to massive financial or reputational repercussions.
Medium	The potential risk is large, but there is some ambiguity surrounding whether or not the issue would practically manifest.
Low	The risk is small, unlikely, or not relevant to the project in a meaningful way.
Code Quality	The issue identified does not pose any obvious risk, but fixing it would improve overall code quality, conform to recommended best practices, and perhaps lead to fewer development issues in the future.

[H-01] Unprotected AssetWrapper Bundle Deposits

~~HIGH~~

AssetWrapper bundles are collections of assets (ETH, ERC721, ERC1155, ERC20). Each type of asset has a corresponding depositX() function. Crucially, each of these functions is open; anyone can deposit an asset into any bundle. The original stated intention of leaving these functions open was to perhaps support gasless transactions for bundle creation in the future.

For an asset to be retrieved, withdraw() must be called. When called, withdraw() performs asset transfers on a loop to the recipient of the assets.

The worst version of this vulnerability occurs when an attacker deposits a ransomware asset–that is, an e.g. ERC1155 asset designed to always revert on transfer, unless the attacker "flips a switch" allowing the transfer to succeed. The attacker then demands a ransom be paid before they'll flip the switch; until the ransom is paid, none of the assets can be retrieved from the bundle.

Both slither and mythril flag this code for review. At the time the vulnerability was brought up by the anonymous security researcher, the following result (and other related matching results) from slither had been set aside for later detailed analysis:

```
AssetWrapper.withdraw(uint256) (contracts/AssetWrapper.sol#145-184)
has external calls inside a loop:
IERC721(erc721Holdings[i_scope_0].tokenAddress).safeTransferFrom(add
ress(this),_msgSender(),erc721Holdings[i_scope_0].tokenId)
(contracts/AssetWrapper.sol#157-161)
AssetWrapper.withdraw(uint256) (contracts/AssetWrapper.sol#145-184)
has external calls inside a loop:
IERC1155(erc1155Holdings[i_scope_1].tokenAddress).safeTransferFrom(a
ddress(this),_msgSender(),erc1155Holdings[i_scope_1].tokenId,erc1155
Holdings[i_scope_1].amount,) (contracts/AssetWrapper.sol#167-173)
```

```
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation/#calls
-inside-a-loop
```

Calling multiple external contracts within a loop, without any try-catch handling, is a precarious proposition in solidity; any contract revert can revert, bricking the function, so any external contract calls made must be considered carefully.

The updated version of AssetWrapper utilizes try-catch, burns any asset that fails during `withdraw()`, and also only allows a bundle owner to deposit assets within a bundle, resolving this issue. The plan for the upcoming V2 is to utilize a different vault-based design.

[L-01] Mismatched return/emit for initializeBundle

~~LOW~~

In AssetWrapper, the _mint() call emits a Transfer event that includes a tokenId. Then, the counter is incremented. Then, the incremented value is returned. As a result, the returned value will not match the emitted value.

The team has informed us that this issue was fixed in a later commit in the repository. For V2, this file will be removed.

[L-02] Griefable FlashRollover

LOW

In flashRollover.sol, rolloverLoan() assumes and checks for an ending balance of 0 of the given payableCurrency. If it does not start with a balance of 0, this check will fail. This opens the flashRollover functionality open to griefing attacks.

To deal with this scenario, Arcade.xyz added a flushToken() function, to manually clear out any tokens that may be present. This is not an ideal mitigation to the issue, however, as a bot could still be implemented that would continuously send dust of the payableCurrency to the contract.

If lender approval were not necessary for flashRollover, as is expected to be the case for V2, this issue would be more serious, as a lender could intentionally attempt to sabotage a rollover to increase their chances of claiming on a defaulted loan.

For V2, the team has informed us that a different mechanism for loan rollovers that does not rely on flash loans will be used.

[L-03] Off-Chain Loan Request Cancelation

LOW

To save gas, loan requests are made as off-chain signed permits. The front end interface also allows canceling these requests. This is done by making this signed permit inaccessible to the lender, presumably by deleting it. There are a number of scenarios where this is insufficient, however–if the deletion does not occur and a copy of the signed data is kept/accessed by the lender, then a previously “canceled” loan request can still be initialized.

To mitigate this, a cancellation nonce on-chain has been discussed for V2.

[L-04] Centralized LoanCore Pause Risk

LOW

AssetWrapper.sol is a contract which maintains ownership of bundles of assets. It is not pausable, and not upgradeable. As the Arcade.xyz team has communicated with us, this is to maintain the trustless nature of a sensitive contract holding ownership of assets. The tradeoff that comes along with this is that trust moves from centralized keyholders to code itself.

LoanCore.sol, however, while not proxy-upgradeable, *is* pausable by Arcade.xyz. During active loans, LoanCore.sol acts as the escrow contract for collateralized bundles of assets. All users of LoanCore.sol are therefore required to trust that no abuse or negligence on behalf of Arcade.xyz will occur.

In a worst case exploit, a holder of `DEFAULT_ADMIN_ROLE` pauses LoanCore while loans are in progress, and demands a ransom. This could happen because of malevolent Arcade.xyz staff, internal or external key compromise, etc. In general, this is expected to be unlikely because (a) Arcade.xyz is presumed to care about its reputation, (b) compromising private keys is hard, when correct private key handling is performed.

However, those presumptions do still require trust. Perhaps the most plausible scenario would be a *fake* key compromise being engineered, wherein the keys are obtained. through some "hack", by an "anonymous black hat", who demands ransom. Likewise, correct private key handling is often not performed, and is by its nature nearly impossible to externally audit. Many serious hacks in the Ethereum ecosystem result from key mishandling. Users must expect and trust that Arcade.xyz performs safe key handling.

The ability of the use of *pause* to be truly helpful compared to the trust it requires is debatable, and all possible solutions involve trade-offs. At a minimum, though, it's recommended that users of Acade.xyz protocol understand the trust they are placing in the Arcade.xyz team as a centralized authority when they have assets worth substantial sums escrowed in contracts they control.

It should be noted that Arcade.xyz does utilize a Gnosis Safe multi-sig for this sensitive role. This can be verified on chain and through the publicly viewable Gnosis Safe UI. At the time of publication, `DEFAULT_ADMIN_ROLE` is held exclusively by a 2/6 multisig (address: 0x398e92C827C5FA0F33F171DC8E20570c5CfF330e), and the Arcade.xyz team has informed us that no single user is capable of pausing the contract.

[Q-01] Use of Deprecated SafeMath

CODE QUALITY

Likely a result of development that started with Solidity < 0.8.0, and later migrated. SafeMath is no longer needed as of Solidity 0.8.0 (see Open Zeppelin), and this contract uses 0.8.0. We have been informed that V2 will not be using SafeMath, and will be using a newer version of solidity.

[Q-02] Missing License Declarations

CODE QUALITY

Solidity conventions recommend including an SPDX License Identifier at the top of every file (see docs). Several source files (FeeController.sol, FlashRollover.sol, et al) in this repository do not contain a license. This also produces noisy compiler warnings, which can make other, more meaningful output from the compiler go unnoticed.

[Q-03] Incorrect Error Message

CODE QUALITY

In `PromisoryNote.sol`, in `_beforeTokenTransfer()`, we see a `require()` check with the error message "`ERC20Pausable: token transfer while paused`"; the token is an ERC721, not an ERC20. This is likely the result of a copy/paste error.

[Q-04] Utilize Checks-Effects-Interactions

CODE QUALITY

In `AssetWrapper.sol`, in `withdraw()`, we see array bundles copied into memory, then external calls made to transfer those assets ("interactions"), and then the storage value is deleted ("effects"). Example:

```
    ERC20Holding[] memory erc20Holdings = bundleERC20Holdings[bundleId];
```

```
        for (uint256 i = 0; i < erc20Holdings.length; i++) {
              IERC20(erc20Holdings[i].tokenAddress).safeTransfer(_msgSender(
              ), erc20Holdings[i].amount);
        }
        delete bundleERC20Holdings[bundleId];
```

While the code in context, in its current state, is not vulnerable, later modifications to the code could cause a re-entrancy vulnerability to be quietly introduced.

Consider using the Checks-Effects-Interactions pattern, executing the delete *before* the external call, like so:

```
        ERC20Holding[] memory erc20Holdings = bundleERC20Holdings[bundleId];
        delete bundleERC20Holdings[bundleId];
        for (uint256 i = 0; i < erc20Holdings.length; i++) {
              IERC20(erc20Holdings[i].tokenAddress).safeTransfer(_msgSender(
              ), erc20Holdings[i].amount);
        }
```

Similarly, in LoanCore, in repay(), we see

```
        require(data.state == LoanLibrary.LoanState.Active,
        "LoanCore::repay: Invalid loan state");
        ...
        IERC20(data.terms.payableCurrency).safeTransferFrom(_msgSender(),
        address(this), returnAmount);
         ...
        loans[loanId].state = LoanLibrary.LoanState.Repaid;
        collateralInUse[data.terms.collateralTokenId] = false;
```

Again, we see Check -> Interaction -> Effect, where we would advise that to proactively defend against the possibility of re-entrancy and conform to best practice, the state change (effect) precede the external call (interaction). This example in particular, when viewed in context, shows how nuanced application of this rule of thumb is required; in theory, setting the LoanState to Repaid immediately after the check would seem to be a good idea (cannot call repay() again if LoanState is Repaid), but if not carefully considered, one could imagine this actually creating other issues with re-entrancy (perhaps later in development, some other functionality becomes available before it should when LoanState is `Repaid`). In more complex cases like this, one might consider using more explicit re-entrancy locks, following patterns found in contracts like Open Zeppelin's Re-Entrancy Guard, Uniswap V2, et al.

Slither flags both of these sections, as well as `LoanCore.startLoan()`, for possible re-entrancy vulnerabilities. Mythril, also, flags many functions for unsafe patterns relating to re-entrancy.

While we are not able to determine any meaningful vulnerability from this code pattern here in the code's current state, and it is clear re-entrancy has been taken into account in contract design, more defensive coding practice is advised for points of higher vulnerability such as these to prevent future hard-to-spot vulnerabilities from quietly appearing during the course of ongoing development.

[Q-05] Copy/Pasted Comments

CODE QUALITY

In IPunks.sol, for example, we see incorrect comments, likely due to copy/paste errors, e.g.:

```
   /**
   * @dev Interface for a permittable ERC721 contract
   * See https://eips.ethereum.org/EIPS/eip-2612[EIP-2612].
   *
   * Adds the {permit} method, which can be used to change an account's
ERC72 allowance (see {IERC721-allowance}) by
   * presenting a message signed by the account. By not relying on
{IERC721-approve}, the token holder account doesn't
   * need to send a transaction, and thus is not required to hold Ether at
all.
   */
```

Note in the comment above where “ERC72 allowance” is typed, when “ERC721 allowance” is likely intended–at least, that’s what was likely intended in IERC721Permit.sol, which is likely the source this was copied from. There are various minor inconsistencies in formatting and potential copy/paste mistakes in comments throughout the repo. Reviewing these for accuracy and correctness is advised.

Gas Optimizations

[G-1] in AssetWrapper, the use of SafeERC20’s safeTransfer function makes sense to enable handling of non-compliant ERC20 tokens–however the use of ERC721 safeTransfer() leads to an unnecessary check to confirm that AssetWrapper itself is a contract is an ERC-721-holder-enabled contract–an unnecessary check that always returns true. Slither catches this, and flags the corresponding code as unreachable.

[G-2] In this design, OriginationController.initializeLoan() is the only function that can trigger incrementing of LoanIdTracker, as well as the _tokenTrackerId of both PromisoryNote instances. Having a single counter and taking loanId to use as the tokenId for the promissory notes directly would remove a significant amount of unnecessary code deployment and execution cost. A further significant savings will also be realized when the redundant checks to `loanIdByNoteId` are then removed, alongside reduced deployment cost.

[G-3] Related to G-2, OriginationController.initializeLoan() is the only function that can call both createLoan() and startLoan(). Those functions seem designed to originally have been called separately, perhaps in a previous design iteration. Because they are always and only called together, and always and only sequentially, some of the operations in createLoan() are unnecessary and can be omitted for gas savings–for example, creating a blank loanTerms template in createLoan() before immediately replacing it with the actual terms in the same call within startLoan().

[G-4] In RepaymentController.repay(), the borrower approves funds to RepaymentController, which then approves that amount to be managed by LoanCore. LoanCore then transfers those funds to itself, and then transfers those funds from itself to the lender. With some minor tweaks, the borrower could approve() LoanCore directly, and LoanCore could transferFrom() borrower to lender directly.

Automated Analysis

Automated tooling is helpful, but fraught with large false-positive rates and takes time to work through, and the results of tools tend to overlap. Generating the most high quality results with the fewest false positives and overlapping results is critical for efficient use of audit time and resources.

Based on research[1] on the topic, we consider a combination of Slither and Mythril to offer the best compromise–data shows that these tools together catch nearly all of the bugs automated tooling can currently catch.

Slither

Slither is a solidity static analysis framework. It detects many vulnerabilities, from high threats to benign ones, of which there are usually many. We ran slither 0.8.1, as well as an updated 0.8.2 (latest at time of audit).

In order to run Slither against the codebase we ran the following command and filtered for relevant files:

- `$ slither .`

Slither flagged many areas for further analysis. While the vast majority of the results were false positives, relevant results have been included where appropriate in the issues above.

Mythril

Mythril is a security analysis tool for EVM bytecode. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.

Mythril is a bit less user friendly. Flattening and compiling the repo is one option, although (1) this can produce misleading results, as flattening is not perfectly equivalent to actual compilation results, and (2) sometimes flattening cannot be done, as in this repository. To produce accurate results for Mythril, we deployed the contracts from this audit onto the rinkeby testnet and executed Mythril tests there.

For some contracts, Mythril will run near-infinitely, so a termination time must be chosen. We ran our Mythril tests on an 11th gen i9-11950H, with an hour per deployed contract address. An example command looked like this:

```
myth analyze -a 0x9a7F9b03aB0C0bA201139C28710f65202CCEC7E2 --rpc
infura-rinkeby --infura-id 0000000000000000000000
--execution-timeout 3600
```

Mythril returned mostly false positives and benign warnings. The overwhelming majority of its results related to the defensive coding paradigm and a lack of utilization of the Checks-Effects-Interactions. There are also warnings about relying on block.timestamp, which do not impact the precision enough in this contract to merit discussion.

Test Coverage

The test coverage is adequate, and test design and organization is overall very good. There is a good mix of unit tests and integration tests covering all major functionality. Coverage was analyzed using Solidity Coverage 0.7.12.

Statements		Branches		Functions		Lines	
97.89%	232/237	88.04%	81/92	90.57%%	48/53	97.92%	235/240

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.